US011163677B2

(12) **United States Patent**
Ostrovsky et al.

(10) **Patent No.: US 11,163,677 B2**
(45) **Date of Patent: *Nov. 2, 2021**

(54) **DYNAMICALLY ALLOCATED THREAD-LOCAL STORAGE**

(71) Applicant: **Microsoft Technology Licensing, LLC**, Redmond, WA (US)

(72) Inventors: **Igor Ostrovsky**, Redmond, WA (US); **Joseph E. Hoag**, Kenmore, WA (US); **Stephen H. Toub**, Seattle, WA (US); **Mike Liddell**, Seattle, WA (US)

(73) Assignee: **Microsoft Technology Licensing, LLC**, Redmond, WA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **16/196,757**

(22) Filed: **Nov. 20, 2018**

(65) **Prior Publication Data**

US 2019/0087316 A1 Mar. 21, 2019

Related U.S. Application Data

(63) Continuation of application No. 15/168,621, filed on May 31, 2016, now Pat. No. 10,133,660, which is a (Continued)

(51) **Int. Cl.**

G06F 12/02	(2006.01)
G06F 9/50	(2006.01)
G06F 3/06	(2006.01)

(52) **U.S. Cl.**

CPC ***G06F 12/023*** (2013.01); ***G06F 3/061*** (2013.01); ***G06F 3/0608*** (2013.01); ***G06F 3/0631*** (2013.01); ***G06F 3/0673*** (2013.01); ***G06F 9/5016*** (2013.01); *G06F 2212/1016* (2013.01); *G06F 2212/1044* (2013.01)

(58) **Field of Classification Search**

CPC G06F 8/443; G06F 9/5026; G06F 12/0804; G06F 12/0817; G06F 12/0815

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,345,313	B1 *	2/2002	Lindholm	G06F 9/5016 719/315
6,820,261	B1 *	11/2004	Bloch	G06F 9/4843 717/114

(Continued)

OTHER PUBLICATIONS

Unknown Author, "Thread Local Storage", msdn.microsoft.com/en-US/library/windows/desktop/ms686749(v=vs.85).aspx, Jul. 1, 2002. (Year: 2002).*

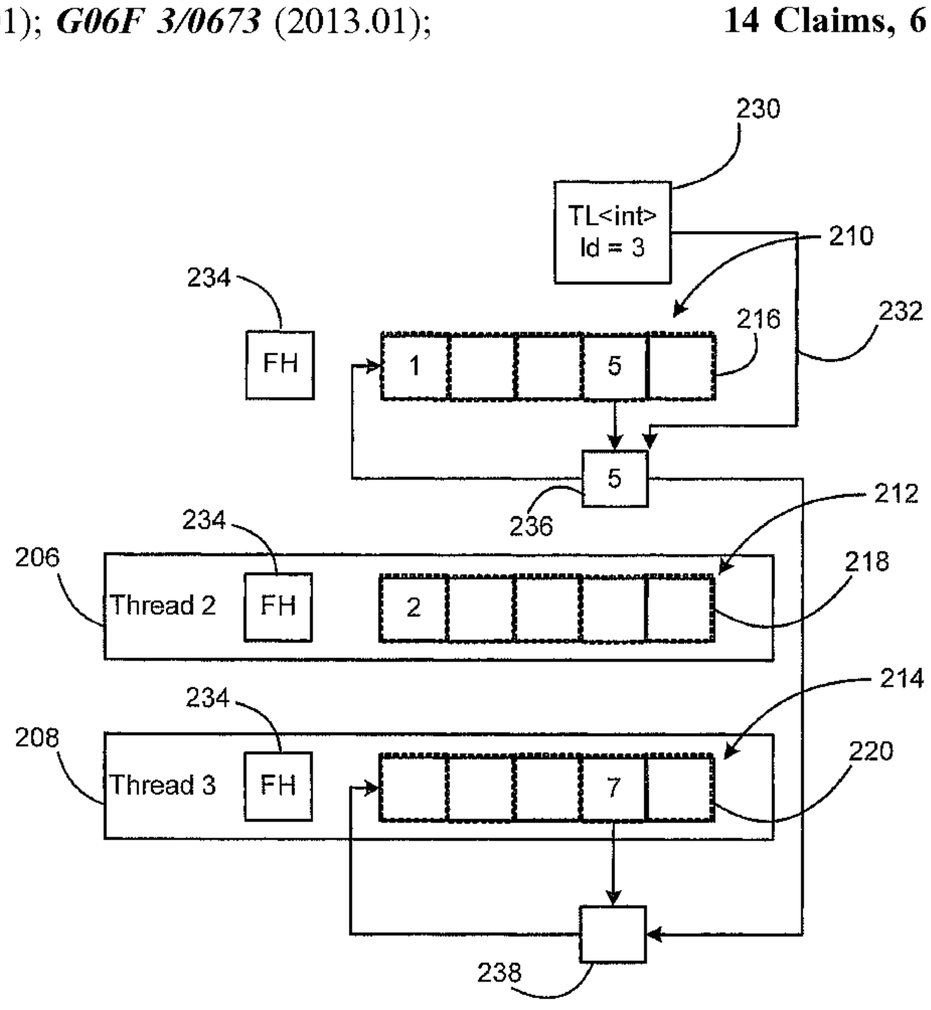
(Continued)

Primary Examiner — Lewis A Bullock, Jr.
Assistant Examiner — Kevin X Lu
(74) *Attorney, Agent, or Firm* — Dicke, Billig & Czaja, PLLC

(57) **ABSTRACT**

Dynamically allocated thread storage in a computing device is disclosed. The dynamically allocated thread storage is configured to work with a process including two or more threads. Each thread includes a statically allocated thread-local slot configured to store a table. Each table is configured to include a table slot corresponding with a dynamically allocated thread-local value. A dynamically allocated thread-local instance corresponds with the table slot.

14 Claims, 6 Drawing Sheets



Related U.S. Application Data

continuation of application No. 13/165,421, filed on Jun. 21, 2011, now Pat. No. 9,354,932.

(56) **References Cited**

U.S. PATENT DOCUMENTS

7,991,962	B2 *	8/2011	Finnie	G06F 9/5016 711/132
2010/0058362	A1 *	3/2010	Cownie	G06F 9/466 719/328
2010/0192139	A1 *	7/2010	Titzer	G06F 9/485 717/151
2011/0126202	A1 *	5/2011	Krauss	G06F 9/485 718/102

OTHER PUBLICATIONS

Unknown Author, "ConcurrentBag <T>.IEnumerable.GetEnumerator Method", msdn.microsoft.com/en-us/library/dd381956(v=vs.100).aspx, Jan. 8, 2007 (Year: 2007).*

Multiple authors, "Algorithm—Grid Data Structure", stackoverflow.com/questions/3544337/grid-data-structure, Aug. 23, 2010 (Year: 2010).*

Unknown Author, "Thread Local Storage 1.35.0—Boost C++ Libraries", Jul. 5, 2008 (Year: 2008).*

Domani et al., "Thread-local heaps for Java", ISMM'02, Jun. 20, 2002 (Year: 2002).*

* cited by examiner

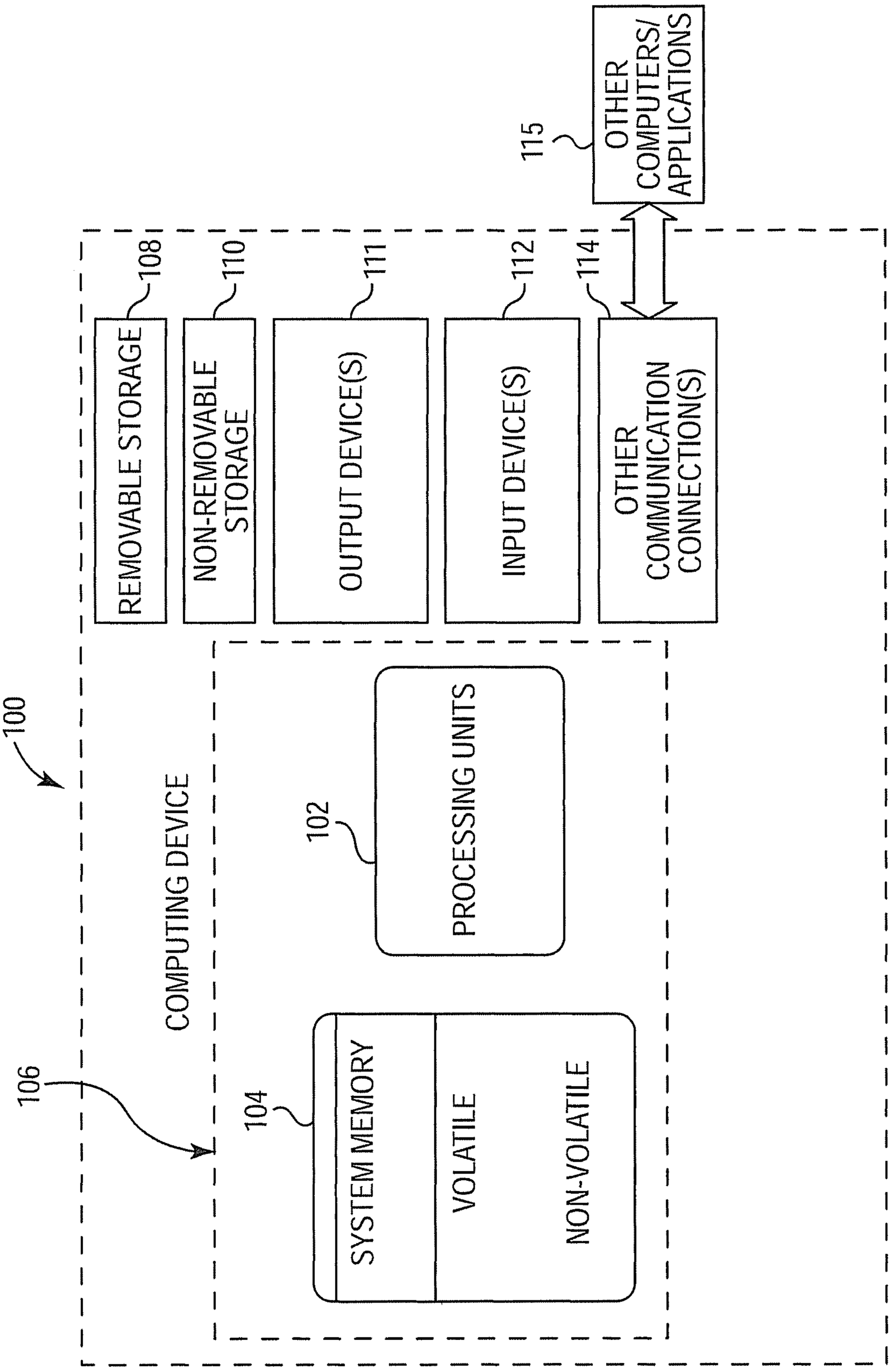

100
COMPUTING DEVICE
102
PROCESSING UNITS
104
SYSTEM MEMORY
VOLATILE
NON-VOLATILE
106
108
REMOVABLE STORAGE
110
NON-REMOVABLE STORAGE
111
OUTPUT DEVICE(S)
112
INPUT DEVICE(S)
114
OTHER COMMUNICATION CONNECTION(S)
115
OTHER COMPUTERS/ APPLICATIONS

Fig. 1

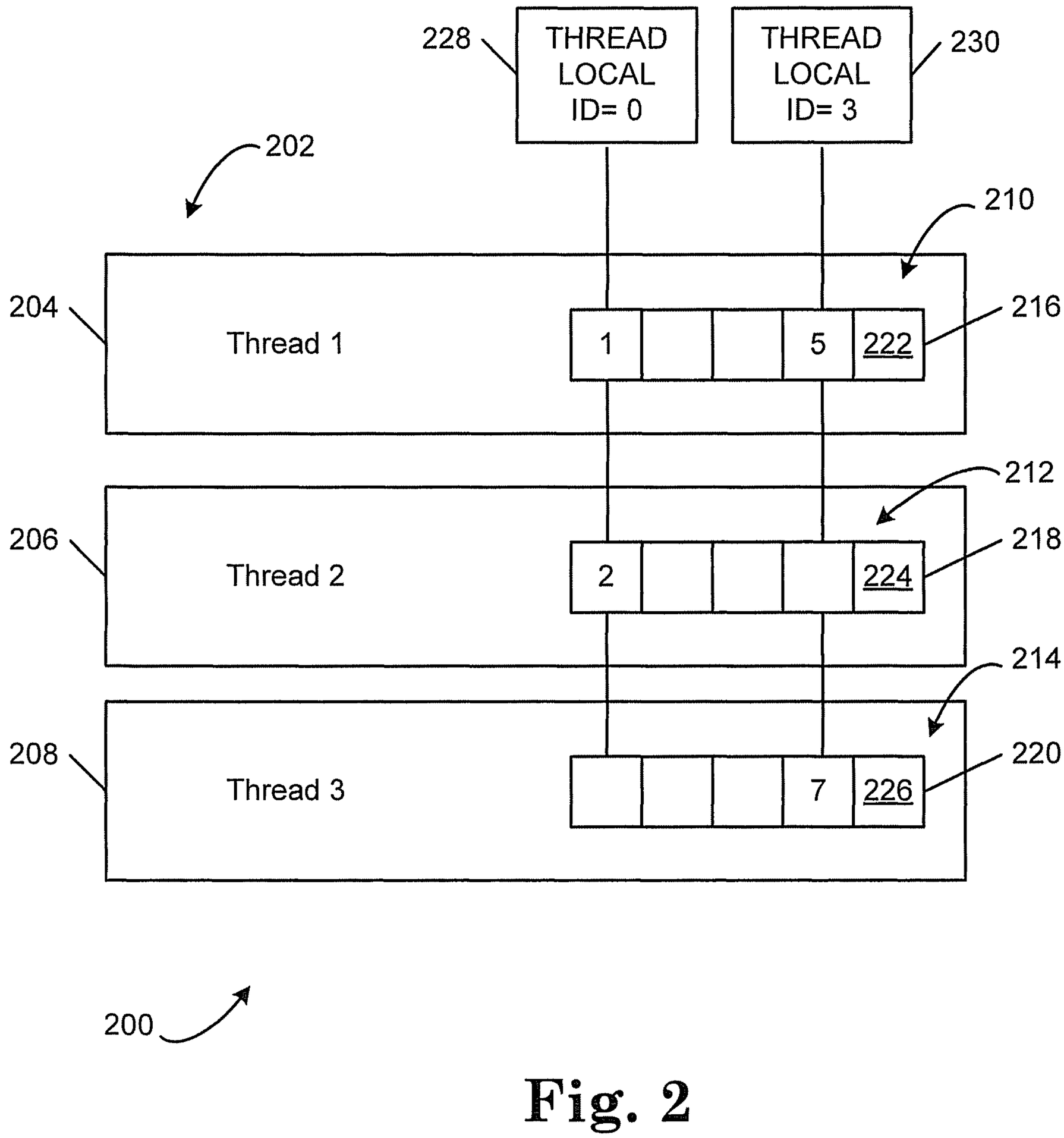

228
THREAD LOCAL ID= 0
THREAD LOCAL ID= 3
230
202
210
204
Thread 1
1
5
222
216
212
206
Thread 2
2
224
218
214
208
Thread 3
7
226
220
200

Fig. 2

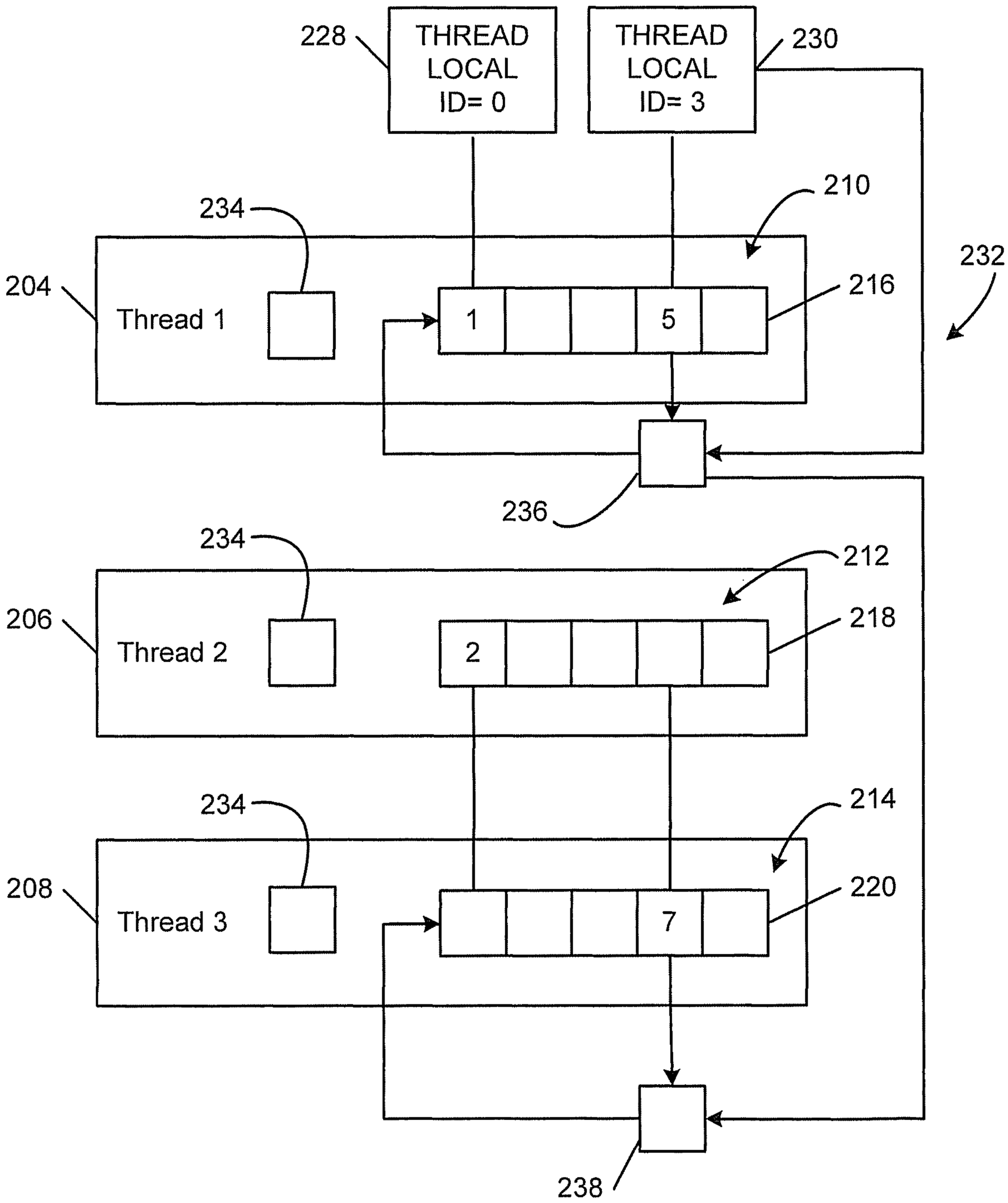

228
THREAD LOCAL ID= 0
THREAD LOCAL ID= 3
230
210
234
232
204
Thread 1
1
5
216
236
234
212
206
Thread 2
2
218
234
214
208
Thread 3
7
220
238

Fig. 3

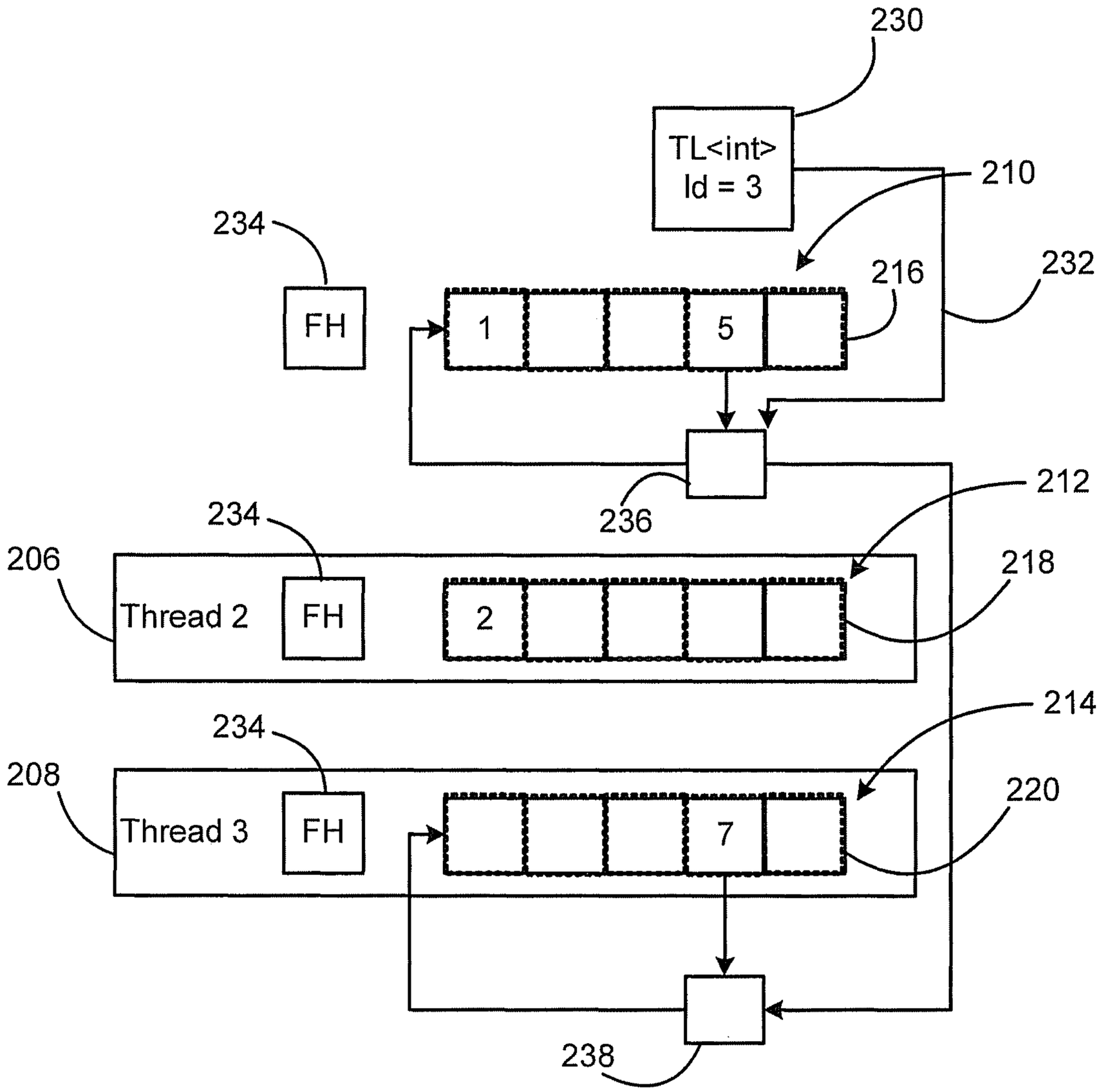

230
TL<int>
Id = 3
210
234
216
232
FH
1
5
236
212
234
206
218
Thread 2
FH
2
214
234
208
220
Thread 3
FH
7
238

Fig. 4A

Fig. 4B

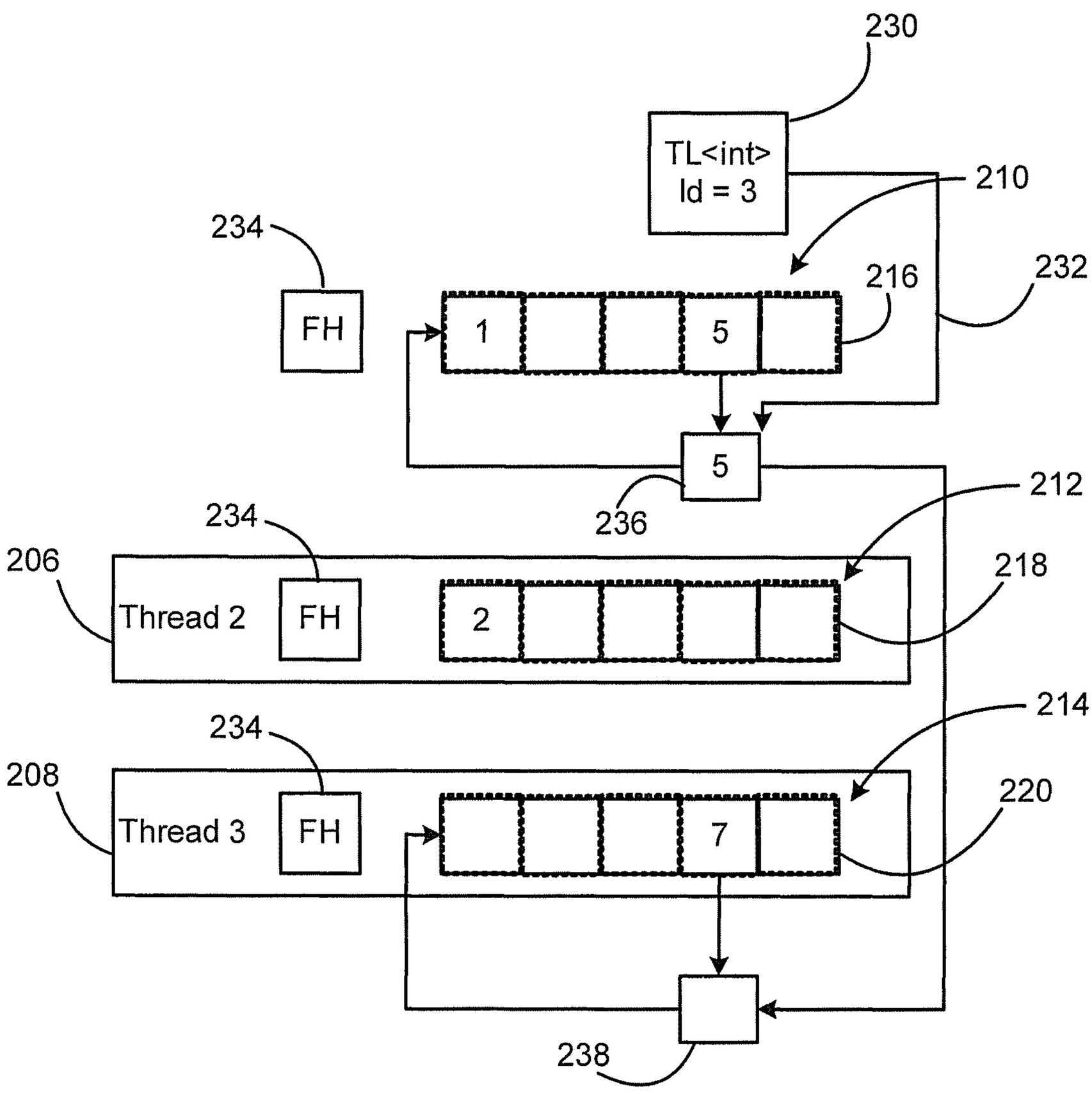
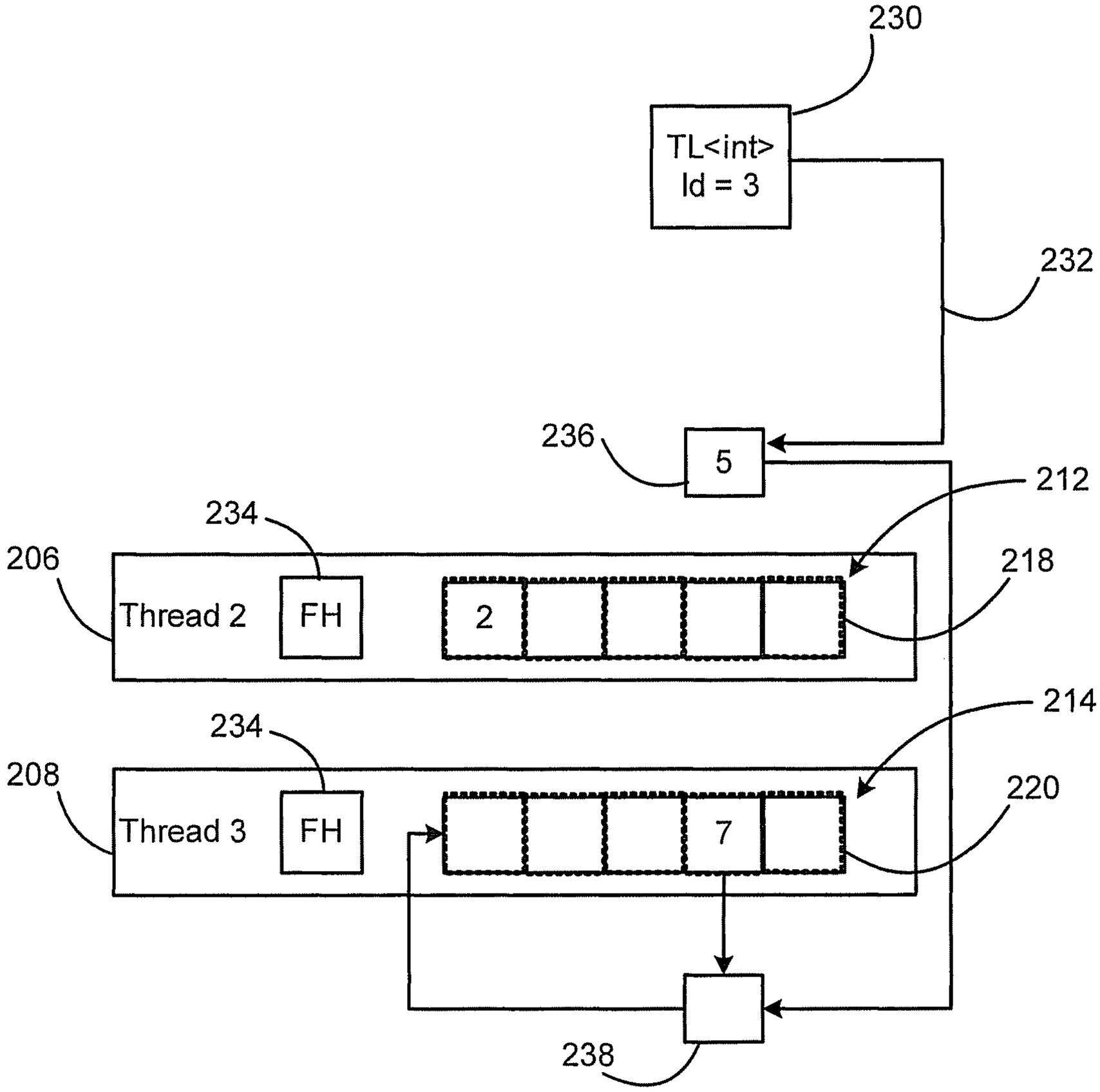

230
TL<int>
Id = 3
210
234
FH
1
5
216
232
5
236
212
234
206
Thread 2
FH
2
218
214
234
208
Thread 3
FH
7
220
238

Fig. 4C


230
TL<int>
Id = 3
232
236
5
212
234
206
Thread 2
FH
2
218
214
234
208
Thread 3
FH
7
220
238

DYNAMICALLY ALLOCATED THREAD-LOCAL STORAGE

CROSS-REFERENCE TO RELATED APPLICATIONS

This patent application is a continuation of Ser. No. 15/168,621 filed May 31, 2016, entitled "DYNAMICALLY ALLOCATED THREAD-LOCAL STORAGE," now U.S. patent Ser. No. 10,133,660, which is a continuation of Ser. No. 13/165,421 filed Jun. 21, 2011, entitled "DYNAMICALLY ALLOCATED THREAD-LOCAL STORAGE," now U.S. Pat. No. 9,354,932, both of which are incorporated herein by reference.

BACKGROUND

Computer applications having concurrent threads executed on multiple processing systems (such as multiple processors, multiple processor cores, or other forms or parallelism) present great promise for increased performance but also present great challenges to developers. The growth of raw sequential processing power has flattened as processor manufacturers have reached roadblocks in providing significant increases to processor clock frequency. Processors continue to evolve, but the current focus for improving processing power is to provide multiple processor cores on a single die to increase processor throughput. Sequential applications, which have previously benefited from increased clock speed, obtain significantly less scaling as the number of processing systems increase. In order to take advantage of multiple processing systems, concurrent (or parallel) applications are written to include concurrent threads distributed over the processing systems.

A process includes one or more threads and the code, data, and other resources of a program in memory. Typical program resources are open files, semaphores, and dynamically allocated memory. A thread is basically a path of execution through a program. A thread typically includes a stack, the state of the processor registers, and an entry in the execution list of the system scheduler. Each thread shares resources of the process. A program executes when the system scheduler gives one of its threads execution control. The scheduler determines which threads will run and when they will run. Threads of lower priority might have to wait while higher priority threads complete their tasks. On multiprocessor machines, the scheduler can move individual threads to different processors to balance the workload. Each thread in a process operates independently. Unless the threads are made visible to each other, the threads execute individually and are unaware of the other threads in a process. Threads sharing common resources, however, coordinate their work by using semaphores or another method of inter-process communication.

Thread Local Storage (TLS) is a method by which each thread in a given multithreaded process can allocate locations in which to store thread-specific data and uses static or global memory local to a thread. Typically all threads in a process share the same address space, which is sometimes undesirable. Data in a static or global variable is typically located at the same memory location, when referred to by threads from the same process. Variables on the stack are local to threads, because each thread has its own stack, residing in a different memory location. Sometimes it is desirable that two threads referring to the same static or global variable are actually referring to different memory locations, thereby making the variable thread-local. If a

memory address sized variable can be made thread-local, arbitrarily sized memory blocks can be made thread-local by allocating such a memory block and storing the memory address of that block in a thread-local variable.

SUMMARY

This summary is provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This summary is not intended to identify key features or essential features of the claimed subject matter, nor is it intended to be used to limit the scope of the claimed subject matter.

Thread-local storage is a programming construct that is useful in multi-threaded programming. A thread-local variable has one memory location for each thread, and each thread will be able to see its own value when accessing the variable. One example scenario where thread-local variables are useful is a two-phase program where each thread first computes a partial result, and then the partial results are combined into a final answer. The thread-local storage can be exposed in a platform with a ThreadStatic attribute. A limitation of ThreadStatic attribute is that each thread-local variable is defined in the program source code and not allocated and released dynamically depending on the input.

The present disclosure is directed to dynamically allocated thread storage in the memory of a computing device. Dynamically allocated thread storage is configured to work with a process including two or more threads. Each thread includes a statically allocated thread-local slot configured to store a table. Each table is configured to include a table slot corresponding with a dynamically allocated thread-local value. A dynamically allocated thread-local instance corresponds with the table slot. Dynamically allocated thread-local slots are implemented on top of statically allocated thread-local slots.

This implementation has several advantages over thread-local variables. For example, the implementation provides for faster reads and writes of thread-local variables. Additionally, the implementation can conserve computer memory. Further, the implementation enables enumeration of the values of all threads for a particular thread-local value.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings are included to provide a further understanding of embodiments and are incorporated in and constitute a part of this specification. The drawings illustrate embodiments and together with the description serve to explain principles of embodiments. Other embodiments and many of the intended advantages of embodiments will be readily appreciated as they become better understood by reference to the following detailed description. The elements of the drawings are not necessarily to scale relative to each other. Like reference numerals designate corresponding similar parts.

FIG. **1** is a block diagram illustrating an example of a computing device for running, hosting, or developing a hash table that can be accessed by two or more concurrent threads.

FIG. **2** is a schematic diagram illustrating an example implementation of dynamically allocated thread-local slots on top of statically allocated thread-local slots configured in the computing device of FIG. **1**.

FIG. **3** is a modified schematic diagram illustrating the example implementation of FIG. **2** during cleanup after a thread-local instance is disposed.

FIGS. **4**A, **4**B, and **4**C are modified schematic diagrams illustrating the example implementation of FIG. **3** in successive stages of a cleanup after a thread exits.

DETAILED DESCRIPTION

In the following Detailed Description, reference is made to the accompanying drawings, which form a part hereof, and in which is shown by way of illustration specific embodiments in which the invention may be practiced. It is to be understood that other embodiments may be utilized and structural or logical changes may be made without departing from the scope of the present invention. The following detailed description, therefore, is not to be taken in a limiting sense, and the scope of the present invention is defined by the appended claims. It is to be understood that features of the various exemplary embodiments described herein may be combined with each other, unless specifically noted otherwise.

FIG. **1** illustrates an exemplary computer system that can be employed in an operating environment such as a distributed computing system or other form of computer network and used to host or run a distributed application included on one or more computer readable storage mediums storing computer executable instructions for controlling a computing device or distributed computing system to perform a method. The computer system can also be used to develop the distributed application and/or provide a serialized description or visualized rendering of the application.

The exemplary computer system includes a computing device, such as computing device **100**. In a basic configuration, computing device **100** typically includes a processor system having one or more processing units, i.e., processors **102**, and memory **104**. Depending on the configuration and type of computing device, memory **104** may be volatile (such as random access memory (RAM)), non-volatile (such as read only memory (ROM), flash memory, etc.), or some combination of the two. This basic configuration is illustrated in FIG. **1** by dashed line **106**. The computing device can take one or more of several forms. Such forms include a person computer, a server, a handheld device, a consumer electronic device (such as a video game console), or other.

Computing device **100** can also have additional features or functionality. For example, computing device **100** may also include additional storage (removable and/or non-removable) including, but not limited to, magnetic or optical disks or solid-state memory, or flash storage devices such as removable storage **108** and non-removable storage **110**. Computer storage media includes volatile and nonvolatile, removable and non-removable media implemented in any suitable method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Memory **104**, removable storage **108** and non-removable storage **110** are all examples of computer storage media. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile discs (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, universal serial bus (USB) flash drive, flash memory card, or other flash storage devices, or any other storage medium that can be used to store the desired information and that can be accessed by computing device **100**. Any such computer storage media may be part of computing device **100**.

Computing device **100** includes one or more communication connections **114** that allow computing device **100** to communicate with other computers/applications **115**. An example communication connection can be an Ethernet interface. In some examples, the computing device can also have one or more additional processors or specialized processors (not shown) to perform processing functions offloaded from the processor **102**. Computing device **100** may also include input device(s) **112**, such as keyboard, pointing device (e.g., mouse), pen, voice input device, touch input device, etc. Computing device **100** may also include output device(s) **111**, such as a display, speakers, printer, or the like.

The computing device **100** can be configured to run an operating system software program and one or more software applications, which make up a system platform. In one example, the computing device **100** includes a software component referred to as a managed, or runtime, environment. The managed environment can be included as part of the operating system or can be included later as a software download. Typically, the managed environment includes pre-coded solutions to common programming problems to aid software developers to create applications, such as software programs, to run in the managed environment. An example of a managed environment can include an application framework or platform available under the trade designation .NET Framework available from Microsoft, Inc. of Redmond, Wash. U.S.A.

The computing device **100** can be coupled to a computer network, which can be classified according to a wide variety of characteristics such as topology, connection method, and scale. A network is a collection of computing devices and possibly other devices interconnected by communications channels that facilitate communications and allows sharing of resources and information among interconnected devices. Examples of computer networks include a local area network, a wide area network, the Internet, or other network.

A computer application configured to execute on the computing device **100** includes at least one process (or task), which is an executing program. Each process provides the resources to execute the program. One or more threads run in the context of the process. A thread is the basic unit to which an operating system allocates time in the processor **102**. The thread is the entity within a process that can be scheduled for execution. Threads of a process can share its virtual address space and system resources. Each thread can include exception handlers, a scheduling priority, thread-local storage, a thread identifier, and a thread context (or thread state) until the thread is scheduled. A thread context includes the thread's set of machine registers, the kernel stack, a thread environmental block, and a user stack in the in the address space of the process corresponding with the thread.

In parallel applications, threads can be concurrently executed on the processor **102**. Concurrent programming for shared-memory multiprocessors can include the ability for multiple threads to access the same data. The shared-memory model is the most commonly deployed method of multithread communication. Multiple threads execute on multiple processors, multiple processor cores, or other classes of parallelism that are attached to a memory shared between the processors.

Thread-local storage is a programming construct that comes useful in multi-threaded programming. An ordinary variable represents a single location in the computer memory, and so all computation threads that access the variable will see and mutate the same value. A thread-local variable, in contrast, has one memory location for each thread, and each thread will see its own value when accessing the variable. One example scenario where thread-local

variables are useful is a two-phase program where each thread first computes a partial result, and then the partial results are combined into a final answer. Such problems can often be conveniently expressed using thread-local storage.

The thread-local storage is exposed in a platform with a ThreadStatic attribute. An example of a ThreadStatic attribute in a platform such as .NET Framework is known as ThreadStaticAttribute. A ThreadStaticAttiribute constructor initializes a ThreadStaticAttribute class in C # (C-sharp) syntax such as: public ThreadStaticAttribute( ). The ThreadStaticAttribute class indicates that the value of a static field is unique for the thread. A static field marked with ThreadStaticAttribute is not shared between threads. Each executing thread has a separate instance of the field, and independently sets and gets values for that field. If the field is accessed on a different thread, it will contain a different value. The ThreadStatic attribute allows the user to annotate a static field as thread-local storage. By using a ThreadStatic attribute, the user can annotate multiple global variables as thread-local. Then, each of those global variables will have one storage slot in each thread in the process, instead of simply having one storage location in total for example.

A limitation of ThreadStatic attribute is that each thread-local variable is defined in the program source code. As a result, thread-local variables are not allocated and released dynamically. Instead, the number of thread-local variables is constant and defined in the program source code.

FIG. **2** illustrates an example implementation **200** of dynamically allocated thread-local slots on top of statically allocated thread-local slots in a computer memory **104** that can be operated on by the processor **102**. The example implementation **200** includes two or more threads **202**, such as three threads **204**, **206**, **208** shown each having a statically allocated thread-local slot **210**, **212**, **214**, respectively. The instance of the statically allocated thread-local slot **210**, **212**, **214**, in each thread **204**, **206**, **208** is configured to store a table **216**, **218**, **220**, respectively. Each thread **204**, **206**, **208** has a different copy of the thread-local slot **210**, **212**, **214** so each thread will include a separate table. Each table **216**, **218**, **220** includes at least one slot **222**, **224**, **226**. In the illustrations, each table **216**, **218**, **220** includes five slots, i.e., slots 0-4. Each slot **222**, **224**, **226** in the table **216**, **218**, **220** can correspond to a dynamically allocated thread-local value.

The example implementation **200** includes at least one dynamically-allocated ThreadLocal instance, such as ThreadLocal instances **228** and **230**. ThreadLocal instance **228** has an identifier (ID) of 0, which indicates that values of ThreadLocal instance **228** are stored in slot, or location, 0 of tables **216**, **218**, **220**. ThreadLocal instance **230** has an ID of 3, which indicates that its values are stored in location **3** of the tables **216**, **218**, **220**. Dynamically allocated thread-local values can be implemented on top of statically allocated thread-local values. A platform such as the .NET Framework exposes GetData and SetData methods for allocating thread-local slots at runtime. The "ThreadLocal class" accomplishes the same and provides a convenient interface.

FIG. **3** is a modified schematic diagram illustrating the example implementation **200** demonstrating an example of ThreadLocal instance **230** being removed. (By modified, FIG. **3** does not include an indication of ThreadLocal instance **228** although it can remain in the implementation **200** or even be the subject of the example.) When ThreadLocal instance **230** is removed, corresponding slots, i.e., location **3**, are removed. On a platform with garbage collection, the value could be a reference to a large object. Until the reference is cleared, i.e., set to a null value, the memory occupied by the large object will not be released even if no other reference to the large object exists.

A cleanup routine is defined for ThreadLocal when ThreadLocal instance, such as ThreadLocal instance **230** is disposed. The cleanup routine can clear out the table slots associated with the disposed ThreadLocal instance **230**. In order to run cleanup routine, however, a linked list **232** is used locate the tables that hold a slot associated with the cleaned up ThreadLocal instance **230**, such as tables **216** and **220**. Linked list **232** is added into the ThreadLocal instance **230** and enumerates over arrays that hold a value for ThreadLocal instance **230**. The linked list **232** thus enables the cleanup routine to locate the tables that hold values for ThreadLocal instance **230**.

While the linked list **232** solves the issue of locating the tables that hold values for the ThreadLocal instance **230**, the linked list **232** by itself also introduces another issue in that the tables **216**, **218**, **220** themselves cannot be garbage collected when threads **204**, **206**, **208** are removed. If the only incoming reference to the tables **216**, **220** is a statically allocated thread-local variable, the thread slots **210**, **212**, **214** can be automatically garbage collected after its owning thread is finished. Now that the thread slots **210**, **212**, **214** are also a part of the linked list **232**, they cannot be garbage collected because the thread slots **210**, **212**, **214** have an additional incoming reference.

To address this issue, the implementation **200** provides for the back-references to a table to be removed when the corresponding thread **202** is removed with a helper cleanup routine **234** that executes when a thread, such as thread **202**, is removed. The helper cleanup routine **234** can be added with a ThreadStatic field in each of the tables **216**, **218**, **220** that holds an object whose cleanup routine performs the desired cleanup that operates to clear all back references to the tables **216**, **218**, **220** when the corresponding thread **204**, **206**, **208** exits.

Thread-local storage can be used to enumerate all values associated with a particular ThreadLocal instance **230**, i.e., the values from all threads. For example, values 5 and 7 are associated with the ThreadLocal instance **230**. The example implementations in this disclosure can extend to support value enumeration.

To save the values for threads that have exited, the helper cleanup routine **233** can be modified after a thread exits. Before releasing the tables **216**, **218**, **220**, the helper cleanup routine **234** saves the final values into nodes **236**, **238** of the linked list **232**. The saved values in the linked list **232** can continue to be included in the enumeration of values for the ThreadLocal instance. FIG. **4** shows the steps in the cleanup after a thread has exited.

FIGS. **4**A to **4**C illustrate an example method of cleanup after a thread has exited from an initial state of the threads **202**, **204** and **206** such as the state illustrated n FIG. **3**. FIG. **4**A illustrates how the linked list **232**, second cleanup object **234** and table **216** remain after thread **204** has exited from the initial state illustrated FIG. **3**. FIG. **4**B illustrates how node **236** of the linked list **232** saves the final value of ThreadLocal instance **230**. The second cleanup routine clears the back references of the table **216**. Afterwards, in FIG. **4**C, the table **216** is garbage collected and the value in node **236** can be enumerated.

Although specific embodiments have been illustrated and described herein, it will be appreciated by those of ordinary skill in the art that a variety of alternate and/or equivalent implementations may be substituted for the specific embodiments shown and described without departing from the

scope of the present invention. This application is intended to cover any adaptations or variations of the specific embodiments discussed herein. Therefore, it is intended that this invention be limited only by the claims and the equivalents thereof.

What is claimed is:

1. A method of allocating thread storage in a memory for processing a plurality of threads, the method comprising:

for each thread of the plurality of threads, generating a statically allocated thread-local slot storing a table, the table having one or more slots wherein each slot position in the one or more slots of the respective thread of the plurality of threads is configured to store a thread-local value associated with a dynamically allocated thread-local instance, the dynamically allocated thread-local instance containing a linked list, each node of the linked list referencing the specific slot position of a table of the plurality of tables with stored values corresponding to the dynamically-allocated thread-local instance;

storing a respective value in a slot of the one or more slots of the table corresponding to the dynamically-allocated thread-local instance in two or more threads of the plurality of threads;

in response to a thread of the two or more threads of the plurality of threads releasing during runtime, performing:

saving the stored thread-local value of the released thread to a respective node of the linked list referencing the table of the released thread;

removing reference of the node of the linked list, to the table of the released thread; and

removing the table and the statically allocated thread-local slot for the released thread of the two or more threads;

wherein, the saved value is accessible for a remaining thread of the two or more threads having a respective value at the specific slot positions of their respective tables.

2. The method of claim **1** comprising:

preserving saved values in the statically allocated thread-local slot for remaining threads of the plurality of threads.

3. The method of claim **1** comprising:

enumerating the saved value associated with a thread-local variable.

4. The method of claim **1** comprising:

enumerating the saved value of the dynamically allocated thread thread-local instance in each of the plurality of threads.

5. The method of claim **1** wherein removing the statically allocated thread-local slot includes locating the table.

6. The method of claim **1** comprising:

defining a cleanup routine for the statically allocated thread-local slot.

7. A computer readable storage medium, which does not include transitory propagating signals, to store computer executable instructions to control a processor to:

for each thread of a plurality of threads, generate a statically allocated thread-local slot storing a table, the table having one or more slots wherein each slot position in the one or more slots of the respective thread of the plurality of threads is configured to store a thread-local value associated with a dynamically allocated thread-local instance, the dynamically allocated thread-local instance containing a linked list, each node of the linked list referencing the specific slot position of a table of the plurality of tables with stored values corresponding to the dynamically-allocated thread-local instance;

store a respective value in a slot of the one or more slots of the table corresponding to the dynamically-allocated thread-local instance in two or more threads of the plurality of threads;

in response to a thread of the two or more threads of the plurality of threads releasing during runtime, perform:

save the stored thread-local value of the released thread to a respective node of the linked list referencing the table of the released thread;

remove reference of the node of the linked list, to the table of the released thread; and

remove the table and the statically allocated thread-local slot for the released thread of the two or more threads;

wherein, the saved value is accessible for a remaining thread of the two or more threads having a respective value at the specific slot positions of their respective tables.

8. The computer readable storage medium of claim **7** wherein the statically allocated thread-local slot is removed with a cleanup routine.

9. The computer readable storage medium of claim **7** wherein the instructions to remove the statically allocated thread local slot includes instructions to garbage collect the statically allocated thread local slot.

10. A system, comprising:

memory to store a set of instructions; and

a processor to execute the set of instructions to:

for each thread of a plurality of threads, generate a statically allocated thread-local slot storing a table, the table having one or more slots wherein each slot position in the one or more slots of the respective thread of the plurality of threads is configured to store a thread-local value associated with a dynamically allocated thread-local instance, the dynamically allocated thread-local instance containing a linked list, each node of the linked list referencing the specific slot position of a table of the plurality of tables with stored values corresponding to the dynamically-allocated thread-local instance;

store a respective value in a slot of the one or more slots of the table corresponding to the dynamically-allocated thread-local instance in two or more threads of the plurality of threads;

in response to a thread of the two or more threads of the plurality of threads releasing during runtime, perform:

save the stored thread-local value of the released thread to a respective node of the linked list referencing the table of the released thread;

remove reference of the node of the linked list, to the table of the released thread; and

remove the table and the statically allocated thread-local slot for the released thread of the two or more threads;

wherein, the saved value is accessible for a remaining thread of the two or more threads having a respective value at the specific slot positions of their respective tables.

11. The system of claim **10** wherein the statically allocated thread-local slot is removed with a cleanup routine to back references to the thread-local slot.

12. The system of claim **11** wherein the cleanup routine is associated with the statically allocated statically allocated thread-local slot of the released thread.

13. The system of claim 11, the processor to execute the set of instructions to perform a garbage collection.

14. The system of claim 10, the processor to execute the set of instructions to enumerate the value associated with the thread-local variable.

* * * * *